Programming Assignment 1
Designing a web crawler

Jaka Kokošar, Danijel Maraž, and Toni Kocjan

Fakulteta za Računalništvo in Informatiko UL dm9929@student.uni-lj.si

Abstract. The article covers the work done in the scope of the first programming assignment as part of the subject web information extraction and retrieval.

Keywords: Data Extraction Retrieval Web Crawler Database Python · Second keyword · Another keyword.

1 Introduction

A web crawler is surely one of the most interesting pieces of software we can write. Mainly because it involves both the world wide web as well as efficient information retrieval and storing with the help of a back end database along with the robustness to handle anything the internet can throw at it. Because of this we decided to implement our crawler in Python which isn't the fastest language but is nice to read and not very verbose along with it's very simple database and web interaction libraries. The crawler like all others makes use of a frontier queue and a very standard Breadth First Search (BFS) strategy with multiple workers simultaneously accessing the frontier, adding pages and parsing them. For the page rendering we utilized the Python Selenium TODO (ali res samo selenium?) library which allowed us to interpret Javascript as most modern pages contain it. Naturally one of the most controversial aspects of a web crawler is whether the page owners impose any restrictions on which pages can be accessed and parsed. To tackle this problem we simply checked inside the robots.txt file which provided us with all the information we needed to know regarding the wishes of the site owners as well as a useful site map that gave us more URLs to add to the frontier. TODO nek zakljucek

2 General structure

Our main file *web_crawler.py* takes as an argument the number of desired worker threads. If no argument is given it defaults to 4. The list *sites* contains the four starting sites as given in the instructions as well as the others chosen by us TODO (zakaj smo izbrali tiste strani). All of the sites are immediately put into our *frontier* queue. The starting arguments are then parsed and the individual worker threads initialized with the *ProcessPoolExecutor* and *submit_worker* functions.

All workers are contained in the list *futures* which with the help of the function *wait* ensures that the program exits when none of the running workers are able to get a new URL from the *frontier* queue. The heart of our crawler is the class Worker which implements all the functionalities of one worker thread. Workers are also supported by a set of global and thread safe data structures.

2.1 Supporting data structures

- The *frontier* queue is the most important supporting data structure. It is implemented using the *Multiprocessing.queue()* class which is thread safe. Workers use it as the main source of new URLs to visit. It contains URLs in a basic canonized *string* form.
- The *visited_dict* dictionary contains URLs in a canonical *string* form that have passed all checks and will be visited in the future or have already been visited. It's main function is to assess whether our crawler has seen this URL before or not. It is implemented with the *multiprocessing.Manager()* dictionary class which is thread safe.
- The *site_domains* dictionary contains key value pairs of domain names and robot parser objects. We use it to check if we've already seen the root domain of the site we're taking from the *frontier*. In case we have we simply get our acquired robots parser object which has already been initialized on the site's robots.txt file. On the contrary if the root domain hasn't been seen before we map it to the corresponding robots parser object. It is implemented with the same class as *visited_dict*.
- The *documents_dict* dictionary contains hashes in *string* form of already parsed html documents. It serves as a very basic mechanism to keep our crawler from parsing sites which have a different URL but the very same contents. It is implemented with the same class as *visited_dict*.

2.2 Worker class

describe general flow here

Init

Call

Get_chrome_driver

Parse_robots

Parse_url

Fetch_url

Parse_page_content

CODE SAMPLE

```
import numpy as np

def incmatrix(genl1,genl2):
    m = len(genl1)
    n = len(genl2)
    M = None #to become the incidence matrix
    VT = np.zeros((n*m,1), int)  #dummy variable

    #compute the bitwise xor matrix
    M1 = bitxormatrix(genl1)
    M2 = np.triu(bitxormatrix(genl2),1)

    for i in range(m-1):
        for j in range(i+1, m):
            [r,c] = np.where(M2 == M1[i,j])
            for k in range(len(r)):
                VT[(i)*n + r[k]] = 1;
                VT[(i)*n + c[k]] = 1;
                VT[(j)*n + r[k]] = 1;
                VT[(j)*n + c[k]] = 1;

                if M is None:
                    M = np.copy(VT)
                else:
                    M = np.concatenate((M, VT), 1)

                VT = np.zeros((n*m,1), int)

    return M
```

3 Database

4 Crawling statistics

5 Conclusion

References

Bonča, J. (2015a). *Sudoku 9x9-15.* Retrieved 2019-01-13, from https://i.pinimg.com/564x/4c/17/f2/4c17f2454b20fa3200882047d1722684.jpg

Bonča, J. (2015b). *Tipkopisi.* Retrieved 2019-01-13, from http://likovnodrustvo-kranj.weebly.com/gostujo269e-razstave/jaka-bonca-tipkopisi

SocialBladeLLC. (2019). *Youtube social blade stats.* Retrieved 2019-01-5, from https://socialblade.com/youtube/

Welbourne, D. J., & Grant, W. J. (2016). Science communication on youtube: Factors that affect channel and video popularity. *Public Understanding of Science*, *25*(6), 706–718.